Forward Integrity and Crash Recovery for Secure Logs

Erik-Oliver Blass[1] Guevara Noubir[2]

[1]Airbus, Munich, Germany
erik-oliver.blass@airbus.com
[2]Northeastern University, Boston-MA, USA
noubir@ccs.neu.edu

***Abstract*—Logging is a key mechanism in the security of computer systems. Beyond supporting important forward security properties, it is critical that logging withstands both failures and intentional tampering to prevent subtle attacks leaving the system in an inconsistent state with inconclusive evidence. We propose new techniques combining forward integrity with crash recovery for secure log data storage. As the support of forward integrity and online nature of logging prevent the use of conventional coding, we propose and analyze a coding scheme resolving these unique design constraints. Specifically, our coding enables forward integrity, online encoding, and most importantly a constant number of operations per encoding. It adds a new log item by XORing it to forward-securely selected k cells of a table. If up to a certain threshold of cells is modified by the adversary, or lost due to a crash, we still guarantee recovery of all stored log items. The main advantage of the coding scheme is its efficiency and compatibility with forward integrity. A key contribution of the paper is the use of spectral graph theory techniques to prove that k is constant in the number n of all log items ever stored and small in practice, e.g., $k = 5$. Moreover, we prove that to cope with up to $\sqrt{n}$ lost log items, storage expansion is asymptotically constant in n and small in practice. For $k = 5$, the total size of the table is only 12% more than the simple concatenation of all n items. We instantiate our scheme into an abstract data structure which allows to either detect adversarial modifications to log items or treat modifications like data loss in a system crash. The data structure can recover lost log items, thereby effectively reverting adversarial modifications.**

I. Introduction

Log services such as `Syslog` collect data about security-relevant events. Logged data is important for security audits and used during forensic analysis, where an analyst investigates how an adversary has attacked a system. Yet, if the adversary is able to fully compromise the machine running the log service, they typically modify stored log data and remove all traces of their attack. As a result, an analyst checking for attacks does not have any means to verify integrity and truthfulness of log data. To cope with compromising adversaries, several previous works have designed mechanisms to store log data with *forward integrity*, see [3, 4, 6, 15, 17–19, 24, 28].

An adversary who has compromised a system has full read-write access to all system data and can therefore easily modify previously logged data items. Roughly speaking, forward integrity ensures that a data item logged at time t is integrity protected such that an adversary compromising the system at time $t' > t$ cannot modify it without being detected. Thus, the goal of forward integrity is not to prevent modifications to data, but to make modifications evident ("tamper evidence") during log analysis. The standard notion of forward integrity is rather simple to achieve. In addition to storing the i^{th} log item $data_i$ in a log file, the log service also stores $\mathsf{HMAC}_{K_i}(data_i)$. Key K_i is evolved to K_{i+1} by computing $K_{i+1} = \mathsf{PRF}_{K_i}(0)$, and K_i is discarded. An adversary compromising the system at time $i+1$ learns K_{i+1}, but not K_i and thus cannot modify previous HMACs without detection. The log service starts with key K_0 which is also known to the analyst (and only to the analyst).

However, a real-world challenge arises from the problem that log files can become inconsistent. Systems crash for various reasons like software bugs, power failure or even hardware failure. As a result, log data is only partially written to disk, previously written data becomes corrupted, and integrity information such as the HMACs does not match. In case of a crash, the analyst would need to accept bad integrity information as a potential crash inconsistency. As demonstrated before [5], an adversary can exploit crashes by performing a crash attack: after compromise, the adversary removes or modifies traces of their attack and then crashes the system. Again, the analyst would accept inconsistent integrity information as a result of the crash, allowing the adversary to evade detection.

A. Contributions

To mitigate such attacks, we propose techniques combining forward integrity with crash recovery for secure data storage. In particular, we present a new a coding scheme with unique design constraints such as forward integrity and most importantly an *online* encoding with a *constant number* of operations per encoded symbol which is not possible with typical error/erasure correction codes such as LDPC, Reed-Solomon, or Fountain codes. In this context, online encoding refers to the property that a symbol is encoded at once and without knowledge of previous encodings.

While the proposed scheme is a random linear code, it has not been considered or analyzed before, as it addresses unique constraints of secure logging. It is not suitable for typical communications and storage scenarios. We instantiate the proposed coding techniques in a new abstract data structure Π with operations $\mathsf{addItem}$ and $\mathsf{listItems}$. Operation $\mathsf{addItem}(data)$ adds data item $data$ (a bit string) to Π, and $\mathsf{listItems}$ outputs all data items in the order they have been previously added. Data structure Π is useful for, e.g., storing a sequence of incoming log entries, but we stress that Π is general, and one can conceive other applications. Besides providing forward integrity and similarly forward confidentiality, the crucial feature of Π is the ability to recover data in case of system crashes with data loss or in case of data corruption.

Specifically, if up to some amount δ of Π's internal data representation gets deleted or corrupted, then $\mathsf{listItems}$ still recovers and outputs all data items with high probability. As δ is a parameter for Π, it is chosen such that it matches the expected amount of data lost during a real-world crash, e.g., due to the system's cache sizes, cache eviction frequency, and file system details. Consequently, the adversary can only modify up to δ data, otherwise malicious modifications become distinguishable from a real crash and lead to detection. Yet, if the adversary modifies at most δ data, $\mathsf{listItems}$ will recover all original data, neutralizing the adversary's modifications.

The key technical challenge in the design of such a data structure is to combine forward integrity and recovery, but still achieve high efficiency in terms of computational complexity for $\mathsf{addItem}$ and $\mathsf{listItems}$ as well as low storage overhead. In many logging scenarios, a log service must be able to cope with a high frequency of incoming log events. Additionally, a log service should also be able to run on embedded devices as in the IoT, where devices are resource constrained, sometimes battery powered, and typically cannot afford high overhead for security features.

Coding Overview: Our coding bears similarity to Gallager's Low-density Parity-Check (LDPC) codes [13, 26]. We, however, have unique constraints that prevent the use of such codes. In order to provide forward integrity and for performance reasons, the encoding has to happen in an online manner without knowledge of previous encodings, and an $\mathsf{addItem}$ operation must only imply a constant number of (simple) computations and disk writes. These requirements *exclude the use of conventional codes* such as LDPC codes [22], or even Reed-Solomon and Fountain Codes (e.g., LT, Tornado, and Raptor) as we discuss in the related work section. Our encoding forward-securely selects k pseudo-random locations in a table and XORs encrypted $data$ to these locations. Thus, we build a system of linear equations, with the table cells representing its right-hand side and indices of pseudo-random locations its left-hand side. We show that if k and the size of the table are chosen appropriately, then the matrix of coefficients of the left-hand side has full rank with high probability $1 - o(1)$. This allows for decodability, i.e., we recover all data using standard Gaussian elimination. More important, even if an amount of up to δ data items in the table becomes invalid, and therewith a certain number of equations are removed from the system, the left-hand side has still full rank and we recover all data.

While the coding scheme is purposefully simple for high efficiency, our main contribution lies in its analysis where we show for which parameters decoding can be guaranteed with high probability. In order to prove decodability guarantees, we extend Calkin's analysis [8] of dependent sets of constant weight binary vectors to binary vectors of hypergeometrically distributed weight. The analysis is of independent interest and leverages spectral graph theoretic techniques connecting the eigenvalues of the transition matrix of a random walk on the hypercube using binary vectors of hypergeometrically distributed weight to the size of dependent set of such vectors. We also show tightness of decodability bounds.

Technical Highlights

- A new online encoding scheme for forward integrity. We formally prove security, decodability, and analyze its complexity. Our proof shows that encoding is extremely efficient and has $\Theta(k)$ time complexity, constant in the number n of data items encoded. Time complexity is not only asymptotically optimal, but also low in practice (e.g., $k = 5$). Computation time is dominated by k applications of cheap symmetric cryptography. Moreover, space overhead is in $\Theta(\frac{1}{1-e^{-k}})$ which is also constant in n and low in practice (12% for $k = 5$). Contrary to related work [14, 21], our coding scheme tolerates not only a constant amount δ of data loss, but any $\delta < \sqrt{n}$.
- We deploy our coding scheme into data structure Π with operations $\mathsf{addItem}$ (performing encoding) and $\mathsf{listItems}$ (performing decoding). Therewith, Π essentially allows ignoring an adversary $\mathcal{A}$ tampering with data. Operation $\mathsf{listItems}$ will still output all data items previously added with probability $1 - o(1)$. As we assume $\mathcal{A}$ to have fully compromised the computer system running the log service, $\mathcal{A}$ can remove more than δ of all content. However in that case, adversary $\mathcal{A}$ is detected with probability 1. $\mathcal{A}$ can not only delete previously added data, but can also modify. However, thanks to forward integrity, $\mathcal{A}$ will be detected in this case with probability $1 - negl(s)$, where s is a security parameter. All data loss due to a regular system crash is recoverable with probability $1 - o(1)$.
- Besides formal analysis, we also implement our techniques and back up our theoretical correctness claims. We use our implementation in millions of experiments and indicate that, as long as $\delta < \sqrt{n}$, $\mathsf{listItems}$ outputs all data items with high probability.

B. Related Work

Secure logging with forward integrity has received some attention, see [3, 4, 6, 15, 17–19, 24, 28] for an overview. However, coping with crash attacks was severely limited so far [5]. An analyst could only distinguish whether an inconsistency in a log file is due to adversarial modifications or to a real-world crash. In contrast, the goal of this paper is to treat data lost in a real crash or modified during a crash attack by using a special encoding of logged data. We treat lost and modified data in the same way and recover up to a configurable amount of δ lost or modified data items. So, we do not just distinguish between a real crash and a crash attack as previous work, but either recover from a real crash, neutralize adversarial modifications or detect the adversary.

There exists previous work on data structures with redundancy which has served as a motivation for this work. Goodrich and Mitzenmacher [14] and Pontarelli et al. [21] store data in a similar fashion as our addItem operation, but later recover by only using a *peeling* mechanism. That is, they check a table of XORs of data item replicas for cells containing only one replica. As long as they find such a cell, they remove the replica from all other cells containing the replica. While peeling (and its analysis) is simple and elegant, it limits the performance to recover all data. In contrast, our rather complex decoding and its analysis show that the encoding allows for high decodability guarantees. Specifically, Goodrich and Mitzenmacher [14] can recover only from a fixed, constant number of lost or modified data items, independently of n, while we support up to $\sqrt{n}$ lost or modified items. In addition, our storage overhead is significantly less: for a similar configuration where a data item is written into $k = 5$ cells in a table, Goodrich and Mitzenmacher [14] require an additional 43% space overhead while we need only 12%. While theoretically possible, none of the previous works provides forward integrity or forward confidentiality as this paper does.

Conventional Coding: At the heart of our approach is a random linear code. Error and erasure correction codes have been extensively studied since the establishment of information theory 1948 and the proof of existence of capacity achieving codes for a variety of channels [9, 25]. Our proposed code bears some similarities to LDPC codes and LT codes. However, it is uniquely restricted by the secure logging requirements, namely that every log data item (symbol) is encoded at once (in an online manner), using a constant number of operations, without maintaining information about the symbols involved in the encoding (to provide forward integrity). While well studied codes such as Reed-Solomon, LDPC, and Fountain codes (LT, Raptor, Tornado) are very efficient in terms of rates and erasure correction ability. Yet, they have constraints that make them unusable in the context of secure logging. In particular, such codes require operations over a non-constant number of symbols for each input symbol, or access to previously coded symbols. For instance, in Reed-Solomon codes, each uncoded input symbol (log data item) contributes to all coded symbols. LDPC codes have sparse parity matrices, but their generator matrices are not sparse. The closest type of codes applicable to this context are Fountain codes, e.g., LT-codes or Raptor codes. While these codes have good performance in typical communications and storage contexts, they cannot be used in this context as they impose requirements on the distribution of the encoded symbols degree (in the bi-partite graph representation), which to the best of our knowledge cannot be achieved due to the forward-security requirements (e.g., Soliton distribution for LT-codes). We acknowledge that, in return, our proposed coding scheme is not suitable for conventional communication and storage scenarios, as secure logging requirements limit its data recovery performance.

II. Background and Adversary Model

In general, data structures with operations are also called Abstract Data Types (ADTs). However, whenever the separation is clear in this paper, we simply refer to Π as a data structure. To allow proper reasoning about security and later data recovery, we briefly formalize both (simple) storage data structures and the threat model.

A. Data Structures for Storage

A storage data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems}, DS)$ comprises state DS and the following three algorithms.

1) $(DS, sk) \leftarrow \mathsf{Init}(1^s, n)$: on input a security parameter s and the maximum number n of data items which will be stored, Init outputs an empty state $DS \in \{0,1\}^{\mathsf{poly}(s)}$. Moreover, Init also outputs auxiliary bit string $sk \in \{0, 1\}^s$. In our specific instance of Π later, sk will be a secret cryptographic key, the start of a key chain to ensure forward integrity and confidentiality.
2) $DS' \leftarrow \mathsf{addItem}(data, DS)$: on input bit string $data \in \{0,1\}^*$, this algorithm adds $data$ to the data structure given by state DS. It outputs an updated state DS'. addItem does not require auxiliary information sk.
3) $(data_1, \ldots, data_\eta) \vee \perp \leftarrow \mathsf{listItems}(DS, sk, n)$: on input a data structure's state DS and auxiliary information sk, listItems outputs *either* a sequence of data $data_i$ *or* special symbol $\perp$ indicating failure. To be able to output failure, e.g., in case of a crash, listItems also receives system parameter n.

DS represents Π's whole state and is, generally speaking, a bit string. In practice, after adding n data *items* to Π with addItem, DS itself is internally organized as a collection of $L(n)$ internal data *values*. For example, a hash table consists of $L(n)$ cells, a tree consists of $L(n)$ nodes etc., but one can imagine various other organizations. Representing Π's whole state, DS does not only contain the collection of internal data values, but might also include cryptographic keys and other data required for operations addItem and listItems.

Any data structure Π for storage must hold two straightforward properties. Informally, if you add $data$, with addItem, then listItems should *with high probability* be able to output $data$ later. Along the same lines, if listItems outputs a sequence of data, then this data should have previously been added with addItem. More formally, we define soundness and completeness (in the absence of crashes or adversarial modifications).

Definition 1 (Soundness). After $(DS_0, sk) \leftarrow \mathsf{Init}(1^s, n)$ and a sequence $(DS_1 \leftarrow (\mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_\eta)), \delta < 1, \eta \leq n$, it holds that

$$Pr[\mathsf{listItems}(DS_\eta, sk, n) = (data_1, \ldots, data_\eta)] = 1 - o(1).$$

Definition 2 (Completeness). If for a state DS_η and auxiliary bit string sk, we have $\mathsf{listItems}(DS_\eta, sk, n) = (data_1, \ldots, data_\eta)$, then

$$Pr[((DS_0, sk) \leftarrow \mathsf{Init}(1^s, n), DS_1 \leftarrow \mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_\eta))] = 1 - o(1).$$

In both definitions, probabilities are taken over random coins of Init, addItem, and listItems.

B. Recovery

A *crash* is an event which modifies or deletes some number δ of Π's internal data values in DS. The exact amount δ can often be estimated in advance as it depends on system parameters such as the system's buffer cache size, physical disk cache size, and cache eviction rates. We now extend soundness and completeness to the case of crashes.

Definition 3 (δ-Recovery). Let DS_η be the result of sequence $((DS_0, sk) \leftarrow \mathsf{Init}(1^s, n), DS_1 \leftarrow \mathsf{addItem}(DS_0, data_1), \ldots, DS_\eta \leftarrow \mathsf{addItem}(DS_{\eta-1}, data_\eta))$, and let DS_η's internal state have $L(n)$ data values. Function $\Delta(DS_\eta, DS'_\eta)$ outputs δ, if state DS'_η is the result of modifying or deleting $\delta \leq L(n)$ internal data values in DS_η.

Storage data structure Π provides δ-recovery, *iff* for all DS_η and DS'_η with $\Delta(DS_\eta, DS'_\eta) \leq \delta$, the calls to $\mathsf{listItems}(DS_\eta, sk, n)$ in definitions 1 and 2 can be replaced by $\mathsf{listItems}(DS'_\eta, sk, n)$, but soundness and completeness still hold with probability $1 - o(1)$.

C. Adversary Model

We now discuss our target security requirements and define an adversary model. We assume that at some point a fully-malicious adversary $\mathcal{A}$ compromises the computer system hosting data structure Π. By compromise, we mean that $\mathcal{A}$ reads out all current memory (RAM, disk) contents and learns all possible cryptographic secrets. This includes the current bit representation DS of Π. Also, $\mathcal{A}$ controls the computer system from now on. That is, $\mathcal{A}$ might diverge from program execution and perform operations of their liking.

Informally, the security properties we want to guarantee are forward Confidentiality and forward Integrity (together abbreviated as *CI*). The notion of forward confidentiality states that $\mathcal{A}$ cannot learn anything about data added to data structure Π before the time of compromise. Forward integrity captures the effect of $\mathcal{A}$'s possible modifications on state DS captured during compromise. Roughly speaking, $\mathcal{A}$'s modifications will not have an effect on the outcome of listItems regarding data items added *before* the time of compromise. That is, listItems will correctly recover all items added before the time of compromise.

As we assume $\mathcal{A}$ to have privileged system access, they can do anything they want with DS, and the strongest integrity notion one can achieve (and we will achieve) is tamper evidence. Such an adversary, allowed to arbitrarily divert from a protocol, but wanting to avoid detection, is also called covert adversaries in the literature [2]. In our case, listItems should *either* output all data added before the time of compromise correctly *or* detect modifications and output $\perp$.

$\{n, (data^0_1, \ldots, data^0_{\eta_0}), (data^1_1, \ldots, data^1_{\eta_1}), st_{\mathcal{A}}\} \leftarrow \mathcal{A}(1^s)$;
// Let $\eta_0, \eta_1 < n$
$\{DS, sk\} \leftarrow \mathsf{Init}(1^s, n)$;
$b \xleftarrow{\$} \{0, 1\}$;
for $i = 1$ **to** η_b **do**
  $DS \leftarrow \mathsf{addItem}(DS, data^b_i)$;
end
$\eta' = \min(n - \eta_0, n - \eta_1)$;
$\{(data'_1, \ldots, data'_{\eta'}), st_{\mathcal{A}}\} \leftarrow \mathcal{A}(st_{\mathcal{A}}, DS)$;
for $i = 1$ **to** η' **do**
  $DS \leftarrow \mathsf{addItem}(DS, data'_i)$;
end
$\{b', DS'\} \leftarrow \mathcal{A}(st_{\mathcal{A}}, DS)$;
confidentiality = False; integrity = False;
if $b \neq b'$ **then**
  confidentiality = True;
end
$result \leftarrow \mathsf{listItems}(DS', sk, n)$;
if $[\mathsf{PREFIX}(result, \eta_b) = (data^b_1, \ldots, data^b_{\eta_b})] \vee [result = \perp \wedge \Delta(DS, DS') > \delta]$ **then**
  integrity = True;
end
output {confidentiality, integrity};

Fig. 1: Experiment $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$

Security Definition: We now formalize our Confidentiality, Integrity, and Recovery intuition. Consider experiment $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$ in Figure 1, where s denotes a security parameter and δ the number of internal data values $\mathcal{A}$ can modify. For a sequence $seq = (data_1, \ldots, data_n)$ of data items, $\mathsf{PREFIX}(seq, \eta)$ outputs the first $\eta \leq n$ items $(data_1, \ldots, data_\eta)$.

In $\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$, adversary $\mathcal{A}$ starts by specifying the maximum number of data items n data structure Π should be able to store. $\mathcal{A}$ outputs two sequences of data items, one with η_0 items, and the other with η_1 items, $\eta_0, \eta_1 \leq n$. One of the two sequences is randomly chosen and added with addItem to an initially empty data structure Π. Then, $\mathcal{A}$ fully compromises the system hosting Π and learns the system's complete state and therewith DS. $\mathcal{A}$ is allowed to add more data items to Π with the constraint that the total number of data items remains less or equal to n. After that, $\mathcal{A}$ again learns DS, can tamper with it in any way they want, and outputs new state DS'. Finally, listItems lists DS' contents.

We require that $\mathcal{A}$ should not have any advantage in breaking confidentiality or integrity of data added before the time of compromise. Note that $\mathcal{A}$ has full access to DS at the time of compromise, but not to the initial auxiliary information

sk generated initially. Only listItems will have access to sk. This setup reflects typical scenarios where, e.g., a logging server adds log entries until eventually another party, the analyst, receives all log entries and analyzes them. In Figure 1, $st_{\mathcal{A}}$ denotes $\mathcal{A}$'s internal state which $\mathcal{A}$ carries through the experiment.

Definition 4. Data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems})$ provides $F(\cdot)$-CIR-security, *iff* for all PPT adversaries $\mathcal{A}$ and same-length data items, there exist function $F(\cdot)$ and negligible function ϵ such that

$$Pr[\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s,\delta).\mathsf{confidentiality} = \mathsf{False}] = \frac{1}{2} + \epsilon(s)$$

and

$$Pr[\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s,\delta).\mathsf{integrity} = \mathsf{False}] \leq F(\cdot),$$

where security parameter s is sufficiently large, and the probabilities are taken over the random coins of $\mathcal{A}$ and Π.

Discussion: The first part of the security definition above addresses forward confidentiality. Even though the adversary gets access to the complete internal state with possible secrets and cryptographic keys, they cannot learn anything about data items added before the time of compromise. $\mathcal{A}$ cannot learn how many data items have already been added. As with standard definitions of confidentiality, e.g., IND-CPA, we require all items $data^0_i, data^1_j$ to have the same length.

The second part of the definition targets integrity and data recovery. If $\mathcal{A}$ has tampered with DS only within up to δ modifications to DS's internal data values, listItems should output all data. If $\mathcal{A}$ has modified more than δ, then listItems outputs $\perp$ indicating failure. In the real world, this corresponds to either recovering all data items inserted or detecting an adversary who modifies more than δ internal values. As we can adjust δ to the expected modifications of a real crash, we will be able to recover from crashes, detect adversarial behavior beyond δ modifications, and cancel adversarial modifications less than δ.

III. Overview of Approach

High-Level Idea: Essentially, we offer a new data structure which allows to add new (log) data items and later list all of them. To add a new data item, we will encode it in a specific way before writing it to disk. The coding we use is customizable, in that it allows listing ("recovering") all previously added data items as long as not more than a threshold parameter δ of the data structure is invalid, e.g., overwritten, corrupted, inconsistent or modified by an adversary. We then set parameter δ to a value which matches the amount of data possibly lost during a real crash of computer system running the logging service. Based on the computer's specifics such as file system cache or disks buffers, one can typically estimate and bound δ. As a result, in case of a real crash, we will be able to recover all previously logged data. Moreover, all adversarial modifications of up to δ data will be neutralized. If the adversary tampers with more than δ, we know that the corrupted data cannot be due to a real crash and therewith provide tamper evidence.

So, our approach has three key components: (1) as its underlying basis, a coding scheme with an online encoding property, i.e., encoding new symbols without knowledge of previous encodings, *constant* write operations per encoded symbol, and high probability of decodability for up to $\sqrt{n}$ erasures, (2) techniques to augment coding schemes to provide forward integrity and forward confidentiality, and (3) an abstract data structure, combining (1) and (2), that provides key secure logging operations of adding and listing logged items while provably guaranteeing properties of forward confidentiality, integrity, and recovery.

In the following, we gradually describe each of the components. After presenting the idea how to encode data and how to provide forward security, we sketch how this is integrated in data structure Π. Formal details with pseudo-code follow in Section IV.

A. *Online Coding Scheme*

Our coding scheme is formally defined through its generator matrix G. An uncoded binary vector $u \in GF(2^n)$ of length n is coded into a codeword $v \in GF(2^m)$ of length m, such that $v = G \cdot u$. Matrix G is an $m \times n$ matrix where each column has exactly k *random* entries equal to 1 and the remaining ones equal to 0. This coding scheme is very efficient to implement and extends to non-binary input symbols u specified one at a time. The idea for encoding is then to XOR each new data item to k distinct, randomly chosen cells in a table of m cells. To decode, one solves the system of linear equations, with the table of m cells defining the right-hand side and G the coefficients of the unknowns of the left-hand side.

One might note similarities to LDPC codes or other efficient erasure codes such as Fountain codes. However, LDPC codes have sparse parity matrices and not generator (encoding) matrices. The computational complexity of encoding LDPC codes is typically not linear in the number of input symbols [22]. While Fountain codes such as Raptor and Tornado codes have linear computational complexity, they cannot encode the input message in an online manner, one input message at a time [20], making them unusable in this context when integrated with forward security mechanisms (see next section). While our code is rather simple, computational complexity of encoding is constant in the number of input symbols and it supports online encoding. Moreover, the intriguing part is its strong decodability guarantee which we will prove.

B. *Forward Security*

One approach to store data with forward confidentiality and integrity builds on key chains, see, e.g., Bellare and Yee [3]. Two parties, the log device and an analyst, initially share a cryptographic key K_0. Both agree that a new key K_i is computed from previous key K_{i-1} by applying, e.g., a PRF $K_i = \mathsf{PRF}_{K_{i-1}}(\gamma)$ for some constant γ or by applying a cryptographic hash function $K_i = h(K_{i-1})$. To store data $item_i$, the log device computes K_i out of K_{i-1}, deletes K_{i-1}, and stores an authenticated encryption of $item_i$ with key K_i. To decrypt and check integrity, the analyst re-computes all K_i starting from K_0. Forward confidentiality and integrity follow from the fact that an adversary can only learn a key K_i at the time of compromise and cannot rewind to previous keys.

Note that in the coding scheme above, to successfully decode, it is required for party B to know G, i.e., the k random locations where each data items was placed in the table. Therefore, the log device derives the k random locations using a PRG, seeded with the current key K_i to encode a symbol. The analyst can then replay random coins and deduce the k locations for each symbol. This is forward secure, because the adversary only learns the current key and cannot replay previously used random locations. Along the same lines, forward security automatically implies that the coding scheme must allow online encoding: information about random locations chosen previously is not available anymore after an encoding.

C. Abstract Data Structure for Forward Secure Storage and Recovery

We now combine coding and forward security techniques and apply them into a new data structure Π. The main idea is to used authenticated encryption with the current key to encrypt the current data item and encode the ciphertext by XORing it to k distinct, pseudo-randomly chosen cells in a table of m cells. After adding n data items we decode: the cells of the table represent the right-hand side of a system of m linear equations, and the cell indices where items were added represent the left-hand side. For certain choices of $m > n$ and k, the resulting system of equations has rank n even if we remove up to δ equations from it. Thus, we can use Gaussian elimination to recover all n data items previously added.

We now give a more technical overview over our data structure Π, with full technical details following in Section IV. Let $k \in \mathbb{N}$ be a small system parameter. Let Algorithms (Enc_K, Dec_K) realize encrypt-then-MAC authenticated encryption in the following sense: the encryption part is IND\$-CPA [23] secure, and integrity tags are generated by a pseudo-random function family. That is, the whole bit string output by Enc_K is indistinguishable from a random string. A standard example for such authenticated encryption is encrypt-then-MAC with AES-CTR and HMAC. Let function PRF_K specify a pseudo-random function family indexed by key K, and $\mathsf{PRG}(s)$ is a pseudo-random generator with seed s.

1) Initialization: First, assume that the maximum number of data items ever to be added into our data structure can be estimated and upper bound by n. Also, assume that all data items have a maximum (padded) length of ℓ bits. During initialization of Π with Init, we create an empty table $\mathcal{T}_1$ with $m = c \cdot n$ cells $\mathcal{T}_1[i]$ for some constant $c > 0$. Cells are indexed by $i \leq m$, and each cell $\mathcal{T}_1[i]$ has length $\ell + O(s)$ bit. Init also outputs an auxiliary information, the start of a key chain $K_1 \xleftarrow{\$} \{0,1\}^s$. The initial state (bit representation) of Π is $DS_1 = (\mathcal{T}_1, K_1)$. So, in our case with a table, the internal data values of our data structure are the table's cells; the number $L(n)$ of internal data values is $L(n) = m = c \cdot n$.

2) Adding data to Π: At the beginning of the i^{th} invocation of $\mathsf{addItem}$, Π's state is $DS_i = (\mathcal{T}_i, K_i)$. We add length ℓ bit item $data_i$ as follows. First, we authentically encrypt $data_i$ to $c_i \leftarrow \mathsf{Enc}_{K_i}(data_i), |c_i| = \ell + O(s)$ bit. Now, instead of hashing c_i to determine where we store it in $\mathcal{T}_i$, we use $\mathsf{PRG}(K_i)$ to randomly choose k distinct locations $(l_1, \ldots, l_k)$ in $\mathcal{T}_i$. We modify table $\mathcal{T}_i$ to become $\mathcal{T}_{i+1}$ by XORing c_i to the k cells with indices l_j. Finally, we compute $K_{i+1} = \mathsf{PRF}_{K_i}(\gamma)$ for some constant γ and set $DS_{i+1} = (\mathcal{T}_{i+1}, K_{i+1})$. Replacing K_i by K_{i+1} allows forward confidentiality and forward integrity. If $\mathcal{A}$ compromises after i data items have been added, they will learn K_i. Thus, $\mathcal{A}$ cannot modify anything that was encrypted with a key K_j for $j \leq i$. Similarly, indices previously generated by PRG are indistinguishable from random bit strings for $\mathcal{A}$.

3) Listing all data items: In contrast, knowledge of K_1 permits deriving all other K_i and therewith cell indices, and we exploit this for $\mathsf{listItems}$. Assume that all n data items have been added to Π. To recover data, $\mathsf{listItems}$ gets K_1 as a parameter. In addition to the key chain of K_i, knowledge of K_1 also permits $\mathsf{listItems}$ to replay all random coins which PRG produced during addition of data items. That is, $\mathsf{listItems}$ knows the indices of where in the table, in which cells, each data item $data_i$ has been XORed to. Recall that XORing equals addition in finite fields of characteristic 2. Therewith, $\mathsf{listItems}$ can set up a system of linear equations $M_n \cdot \vec{x} = \mathcal{T}_n$ where M_n is a $m \times n$ matrix over $GF(2)$ and table $\mathcal{T}_n$ can be parsed as a length m vector over $GF(2^{\ell+O(s)})$. Each cell of the table represents one component of the vector. We show the augmented matrix $AM = [M_n|\mathcal{T}_n]$ in Figure 2. Each column vector of M_n specifies where ciphertext c_i has been XORed to $\mathcal{T}_n$. Note that each column vector has constant weight k.

$$AM = \begin{array}{c} \begin{array}{cccc|c} c_1 & c_2 & \cdots & c_n & \mathcal{T}_n \end{array} \\ \left[\begin{array}{cccc|c} 0 & 0 & \cdots & & \mathcal{T}_n[1] \\ \vdots & 1 & \vdots & & \vdots \\ 0 & 0 & & & \\ 1 & \vdots & & & \\ 0 & 0 & & & \\ \vdots & 1 & & & \\ 0 & 0 & & & \\ 1 & \vdots & & & \\ 0 & & & & \vdots \\ \vdots & & & & \mathcal{T}_n[m] \end{array}\right] \end{array}$$

Fig. 2: Augmented matrix $AM = [M_n|\mathcal{T}_n]$

Algorithm $\mathsf{listItems}$ now solves the system of linear equations AM over $GF(2^{\ell+O(s)})$ to receive candidates for c_i.

Then, it verifies the integrity tag for each candidate c_i and outputs c_i if verification succeeds. Remember that listItems can re-compute all K_i used during authenticated encryption due to knowledge of K_1. To solve the system of linear equations, we use Gaussian elimination. The key observation is that as long as M_n has rank n, Gaussian elimination will always succeed and output sequence $(c_1, \ldots, c_n)$. We will prove in Section V that M_n has rank n with high probability, even if δ equations are removed from M_n and $\mathcal{T}_n$. Our proof extends results from Calkin [8] on the rank of sets of independent, constant weight binary vectors, to hypergeometrically distributed weight (due to data loss).

Note that there are several optimizations possible which we discuss later in Section V. We will also discuss the exact choice of parameters c and k there.

4) Caveats: Before we conclude our high level overview and proceed with technical details, we emphasize that there are several additional techniques required to make this approach really secure.

- First, there must be a way for listItems to verify whether the i^{h} cell of the table $\mathcal{T}_j$ is broken, i.e., whether a crash has overwritten parts of the cell's content. Otherwise, a broken bit sequence in $\mathcal{T}_j[i]$ will lead to an inconsistent system of linear equations. To mitigate, we extend each cell in the table by another integrity tag T_i. So, each cell comprises as its first part XOR_i the XOR of authenticated encryptions and as a second part an integrity tag T_i. During insertion of $data_j$, after addItem has XORed c_j to first part XOR_i of cell $\mathcal{T}_j[i]$, we write $\mathsf{HMAC}_{K_j}(\mathsf{XOR}_i)$ into the second part T_i of $\mathcal{T}_j[i]$. Later, algorithm listItems will remove all equations from AM where the HMAC part does not match the XOR_i part. Adversary $\mathcal{A}$ can arbitrarily modify contents of cell $\mathcal{T}_j[i]$ and then compute a T_i using their current K_j. However, if $\mathcal{A}$ introduces inconsistencies in AM, listItems knows that such inconsistencies cannot be a result of a crash. So, listItems will output $\perp$ upon observing an inconsistency.
  Finally, for listItems to understand which key K_j was used to generate tag T_i, we also store $ID_i = \mathsf{PRF}_{K_j}(\gamma')$ in $\mathcal{T}_j[i]$, where γ' is another constant different from γ. Therewith, listItems can once generate all n possible IDs, store them in a separate hash table, and then access them in expected time $O(1)$. In conclusion, each cell $\mathcal{T}_j[i]$ comprises XOR_i, T_i, and ID_i.
- Also, $\mathcal{T}_1$ cannot be empty (e.g., filled with zeros) in the beginning. Otherwise, $\mathcal{A}$ would be able to distinguish between empty and non-empty cells in $\mathcal{T}_n$ with some probability. A simple strategy for $\mathcal{A}$ would then be to focus on deleting non-empty cells from the table. We could not guarantee forward integrity with high probability.
  Consequently, we initialize $\mathcal{T}_1$ not with zeros, but fill it pseudo-randomly. For example, we start with a random K_0 and fill $\mathcal{T}_1$ with the output of $\mathsf{PRG}(K_0)$. Later, listItems can remove the initial pseudo-randomness by re-computing it and XORing it to $\mathcal{T}_n$.

// Let $c > 1, k > 2$ be constant system parameters, $\gamma \xleftarrow{\$} \{0,1\}^s$ a constant
$m = c \cdot n$;
// Generate table of m cells, each of size $\ell + 4 \cdot s$ bit: $\ell + 2 \cdot s$ bit for encrypted data item (including Enc's random coins and integrity tag), and $2 \cdot s$ bit for cell authentication T_j and key identifier ID_j
$\mathcal{T}_0 = \mathsf{EmptyTable}(m, \ell + 4 \cdot s)$;
$K_0 \xleftarrow{\$} \{0,1\}^s$;
// Generate $m \cdot (\ell + 4 \cdot s)$ bit pseudo-random pad
$pad = \mathsf{PRG}(K_0)$;
$\mathcal{T}_1 = \mathcal{T}_0 \oplus pad$;
$K_1 = \mathsf{PRF}_{K_0}(\gamma)$; $DS_1 = (\mathcal{T}_1, K_1)$;
output (DS_1, K_0);

Algorithm 1: $\mathsf{Init}(1^s, n)$

// Parse DS_i as $(\mathcal{T}_i, K_i)$
$c_i \leftarrow \mathsf{Enc}_{K_i}(data_i)$;
// Generate k Distinct Random Numbers (DRN) between 1 and $m = c \cdot n$)
$\{l_1, \ldots, l_k\} \leftarrow \mathsf{DRN}(\mathsf{PRG}(K_i))$;
// Compute $\mathcal{T}_{i+1}$ out of $\mathcal{T}_i$
$\mathcal{T}_{i+1} = \mathcal{T}_i$;
foreach l_j **do**
  // Parse $\mathcal{T}_{i+1}[l_j]$ as $(\mathsf{XOR}_{l_j}, T_{l_j}, ID_{l_j})$, let $\gamma' \xleftarrow{\$} \{0,1\}^s$ be a constant
  $\mathsf{XOR}_{l_j} = \mathsf{XOR}_{l_j} \oplus c_i$;
  $T_{l_j} = \mathsf{HMAC}_{K_i}(\mathsf{XOR}_{l_j})$;
  $ID_{l_j} = \mathsf{PRF}_{K_i}(\gamma', j)$;
end
$K_{i+1} = \mathsf{PRF}_{K_i}(\gamma)$;
output $DS_{i+1} = (\mathcal{T}_{i+1}, K_{i+1})$

Algorithm 2: $\mathsf{addItem}(data_i, DS_i)$

IV. Detailed Description

We now turn to technical details and describe Π with both coding as well as forward confidentiality and integrity techniques.

A. Init

For an estimated upper bound of n data items, Algorithm 1 (Init) generates table $\mathcal{T}$ with m cells, where $m = c \cdot n, c > 1$. Init also generates a key K_0. The total size of each cell in $\mathcal{T}$ is $\ell + 4 \cdot s$ bit. The first $\ell + 2 \cdot s$ bit are reserved for the authenticated encryption of a size ℓ bit data item and include the encryption's random coins and integrity tag (e.g., AES-CTR and HMAC). From now on, we will call the first $\ell + 2 \cdot s$ bit of the cell the XOR part, as this is what will be XORed during addItem. Each cell also includes s bit for an additional integrity tag T

(e.g., HMAC), and another s bit for a so called key ID (a PRF output which we will explain below). In conclusion, the i^{th} cell $\mathcal{T}[i]$ comprises $(\mathsf{XOR}_i, T_i, ID_i)$.

Init fills each cell with the output of $\mathsf{PRG}(K_0)$, see Line 5. Finally, Init outputs Π's state DS which is the table, the next key K_1, and the number of data items in Π, i.e., 0.

B. addItem

To add a new data item $data_i$ to data structure Π, which already holds $i-1$ data items, Algorithm 2 (addItem) first authentically encrypts $data_i$ with current key K_i to ciphertext c_i. Then, addItem pseudo-randomly chooses k distinct indices l_j in table $\mathcal{T}$. To generate required (pseudo-)randomness, addItem uses pseudo-random generator PRG with K_i as seed. For each cell $\mathcal{T}[l_j]$ indexed by l_j, addItem XORs c_i to $\mathcal{T}[l_j]$'s XOR part. It then computes HMAC_{K_i} over this XOR part and stores the result as integrity tag T_{l_j} in $\mathcal{T}[l_j]$. Finally, to later help listItems to find correct key K_i for decryption, addItem also stores key ID $ID_{l_j} = \mathsf{PRF}_{K_i}(\gamma', j)$ in $\mathcal{T}[l_j]$. Here, "," is an unambiguous pairing of inputs such as concatenation of fixed-length j and constant γ'. Operation addItem outputs the updated table and a new key K_{i+1}.

To be able to IND\$-CPA encrypt, we require all data items to have the same length ℓ. In practice, data items might therefore require padding.

C. listItems

Algorithm 3 (listItems) recovers all data items from DS. It receives initial key K_0 as a parameter and starts by re-computing all n possible keys K_i. To be able to link key K_i with its k corresponding IDs, listItems uses a simple (key,value) store KeyStore. For all of the k possible locations l_j where K_i could have been used to compute T_{l_j}, it stores tuple (K_i, i, l_j) under its key ID in KeyStore.

The main idea now is to compute $m \times n$ matrix M, the matrix of coefficients over $GF(2)$ representing the left-hand side of the system of linear equations. M is initially all zero. As a first step, listItems computes the number of log entries which have been added to Π, and therewith M's expected *rank*, by checking which key IDs and therewith which keys have been used (Line 13).

Then, using keys K_i, listItems replays all indices for item $data_i$. For each position l_j in $\mathcal{T}$, listItems puts a 1 in column i, row l_j of M. However, it does this only, if the corresponding integrity tag T_{l_j} matches XOR_{l_j}, i.e., if this table cell was not modified by the adversary or during a crash. To check T_{l_j}, listItems fetches the corresponding key from KeyStore using ID_{l_j}. Note that listItems also verifies whether content in $\mathcal{T}[l_j]$ is supposed to be at position l_j (Line 20). If one of the checks fails, coefficient $M[l_j, i]$ remains 0, and XOR_i is set to $0^{\ell+2\cdot s}$. Therewith, we essentially remove this equation from the system of equations. To remove the initial randomness from all XOR_i, listItems re-computes the random bit string pad and XORs it to all remaining (non-zero) cells in $\mathcal{T}$. We represent resulting $\mathcal{T}$ as a vector $\vec{v}$ of dimension m.

// Parse DS as $(\mathcal{T}, K_\eta)$ and $\mathcal{T}[i]$ as $(\mathsf{XOR}_i, T_i, ID_i), 1 \le i \le m$
// Let KeyStore be a (key,value)-pair data structure.
// Generate all possible keys and IDs, store in KeyStore.
1 **for** $i = 1$ **to** n **do**
2 $\quad K_i = \mathsf{PRF}_{K_{i-1}}(\gamma)$;
$\quad$ // Re-Generate k Distinct Random Numbers (DRN)
3 $\quad \{l_1, \ldots, l_k\} \leftarrow \mathsf{DRN}(\mathsf{PRG}(K_i))$;
4 $\quad$ **for** $j = 1$ **to** k **do**
5 $\quad\quad ID = \mathsf{PRF}_{K_i}(\gamma', j)$;
6 $\quad\quad \mathsf{KeyStore.put}(ID, (K_i, i, l_j))$;
7 $\quad$ **end**
8 **end**
9 M=$m \times n$ zero matrix over GF(2);
// Predict M's rank $rank$
10 $rank = 0$;
11 **for** $i = 1$ **to** m **do**
12 $\quad (K_j, j, \ell) = \mathsf{KeyStore.get}(ID_i)$;
13 $\quad rank = \max(rank, j)$;
14 **end**
15 **if** $rank = 0$ **then output** $\perp$; **end**
16 **for** $i = 1$ **to** $rank$ **do**
17 $\quad \{l_1, \ldots, l_k\} \leftarrow \mathsf{DRN}(\mathsf{PRG}(K_i))$;
18 $\quad$ **for** $j = 1$ **to** k **do**
19 $\quad\quad (K_t, t, l) = \mathsf{KeyStore.get}(ID_{l_j})$;
20 $\quad\quad$ **if** $l = l_j$ **and** $\mathsf{HMAC}_{K_t}(\mathsf{XOR}_{l_j}) = T_{l_j}$ **then** $M[l_j, i] = 1$; **else** $\mathsf{XOR}_{l_j} = 0^{\ell+2\cdot s}$; **end**
21 $\quad$ **end**
22 **end**
// Re-generate $m \cdot (\ell + 4 \cdot s)$ bit pseudo-random pad
23 $pad = \mathsf{PRG}(K_0)$; // Let $pad[i]$ denote the $\ell + 2 \cdot s$ bit string covering XOR_i in $\mathcal{T}[i]$.
24 **for** $i = 1$ **to** m **do**
25 $\quad$ **if** $\mathsf{XOR}_i \neq 0^{\ell+2\cdot s}$ **then** $\mathsf{XOR}_i = \mathsf{XOR}_i \oplus pad[i]$ **end**
26 **end**
// Let vector $\vec{v} = (\mathsf{XOR}_1, \ldots, \mathsf{XOR}_m) \in GF(2^{(\ell+2\cdot s)^m})$
// Let vector $\vec{c}$ be a solution vector over $(GF(2^{\ell+2\cdot s}))^n$
27 Solve $M \cdot \vec{c} = \vec{v}$ for $\vec{c}$;
// Let $c_1, \ldots, c_{rank}$ be the unique solutions
28 **for** $i = 1$ **to** $rank$ **do**
29 $\quad data_i = \mathsf{Dec}_{K_i}(\vec{c}[i])$; // Decryption might fail
30 $\quad$ **if** $data_i = \perp$ **then output** $\perp$; **end**
31 **end**
32 **output** $(data_1, \ldots, data_{rank})$;

Algorithm 3: $\mathsf{listItems}(DS, K_0, n)$

Finally, $\mathsf{listItems}$ solves the resulting system of linear equations $M \cdot \vec{c} = \vec{v}$, e.g., by applying Gaussian elimination. This results in solution $\vec{c}$ which $\mathsf{listItems}$ decrypts componentwise to get $data_i$. If Gaussian elimination finds an overdetermined system of equations, or if integrity does not check during decryption, $\mathsf{listItems}$ outputs $\perp$ (Line 30). Otherwise, it outputs $(data_1, \ldots, data_{rank})$.

D. Additional Security Details

To keep our exposition clear, we have omitted two important security properties.

Minimum Rank: First, we need a minimum rank. Note that $\mathsf{listItems}$ of Algorithm 3 would output $\perp$ for a freshly initialized data structure. That is, $(DS, K) \leftarrow \mathsf{Init}(1^s, n)$, $\mathsf{listItems}(DS, K, n)$ returns $\perp$. Moreover, during compromise, adversary $\mathcal{A}$ can change state DS to a garbage state DS' by overwriting DS with random bit strings. A completely random DS' violates the requirement of up to δ modifications or deletions. As $\mathsf{listItems}$ will not find any valid ID, it would set $rank$ to 0, M would remain a $m \times n$ zero matrix, and $\mathsf{listItems}$ would output an empty set, but not $\perp$.

However, we remedy this issue by simply adding a single dummy data item during initialization. We create a table for $n+1$ data items and then call $\mathsf{addItem}$ once to add a dummy data item. If $\mathsf{listItems}$ detects a rank of 0, it outputs $\perp$.

Separate Keys: Second, in the presentation above we use the same key K_i as a key for different cryptographic primitives $\mathsf{Enc}, \mathsf{HMAC}, \mathsf{PRF}$, and PRG. While this keeps the exposition simple, it leads to unclear security. Instead, one would use different keys for each cryptographic primitive, i.e., K_i^{X} for primitive $\mathsf{X} \in \{\mathsf{Enc}, \mathsf{HMAC}, \mathsf{PRF}, \mathsf{PRG}\}$. In Line 9 of Algorithm 2, each of these keys will then be evolved by computing $K_{i+1}^{\mathsf{X}} = \mathsf{PRF}_{K_i^{\mathsf{X}}}(\gamma)$. An interesting alternative would be to each time derive key K_i^{X} for primitive X from key K_i using a derivation function such as $K_i^{\mathsf{X}} = \mathsf{PRF}_{K_i}(\gamma_{\mathsf{X}})$ and different constants γ_{X}. Only K_i would be evolved to K_{i+1}.

V. Analysis

In this section, we prove that Π is a data structure providing $o(1)$-CIR-security. That is, for function F from Definition 4 we have $F(n) = o(1)$.

As our key chain techniques for forward integrity and confidentiality are rather standard, we dismiss adversaries $\mathcal{A}$ violating confidentiality ($\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$.confidentiality is True) or integrity ($\mathsf{Exp}^{\mathsf{CIR}}_{\mathcal{A},\Pi}(s, \delta)$.integrity is True) by breaking cryptographic primitives. Let s be the security parameter for cryptographic primitives $\mathsf{PRG}, \mathsf{PRF}$, and Enc used in their respective security definitions. We summarize that the probability of $\mathcal{A}$ breaking forward integrity and confidentiality by attacks on cryptographic primitives is negligible in s. Also note that $\epsilon(s) \in o(1)$.

The focus of our analysis is to formally prove that integrity can also not be violated in case of a crash or an adversary tampering with up to δ data items, i.e., δ-recovery. As a warm-up to this, we first prove that, without a crash or adversarial behavior, Π is both sound and complete.

A. Soundness and Completeness

Lemma 1. For any $k \geq 2$, there is a $c \in O(1)$ such that data structure $\Pi = (\mathsf{Init}, \mathsf{addItem}, \mathsf{listItems}, DS)$ is sound and complete with probability $1 - o(1)$.

Proof. The key insight is that our way of constructing matrix M corresponds to adding random binary length m vectors with exactly k "1"s to an initially empty set S. There exist several fundamental results quantifying a threshold size n of S, $n = |S|$, until which vectors remain linearly independent and starting from which vectors in S become linearly dependent with high probability [7, 8, 10, 11].

For $k = 2$, if $m = c \cdot n$ and $c > 2$, then the n vectors in S remain linearly independent almost surely with increasing m, n, i.e., with high probability $1 - o(1)$ [11].

For each $k \geq 3$, it has been shown that there exists a c such that, as long as $n < m/c$, the n vectors in S remain linearly independent with probability $1-o(1)$, see Theorem 2b in Calkin [8]. Moreover, c can be approximated by $c^{-1} \approx 1 - \frac{e^{-k}}{\ln 2} - \frac{e^{-2 \cdot k}}{2 \cdot \ln 2} \cdot (k^2 - 2 \cdot k + \frac{2 \cdot k}{\ln 2} - 1)$. Note that, with increasing k, values for c go (exponentially fast) towards 1.

A linear independent set S directly translates to our $m \times n$ matrix M having rank n. If M's rank is n, $\mathsf{listItems}$ solves the related system of linear equations with probability 1. If $\mathsf{listItems}$ can solve the system of linear equations, soundness (Definition 1) and completeness (Definition 2) follow immediately. □

On a side note, we remark that for $k = 1$ the n vectors in S remain linear independent with probability $1 - o(1)$ only if $n < \sqrt{m}$. This is a well known result used, e.g., in the theory of constructing perfect hash functions. However, it also implies that there is no c which is constant in n and which would result in a rank n matrix M with our $\mathsf{addItem}$ procedure. In conclusion, we have to set $k \geq 2$ for Π.

For the remainder of the paper, we set $k \geq 3$ as it allows for tailoring storage requirements and computational overhead. For example, Dietzfelbinger and Pagh [10] present that for $k = 5$ we must set $c > 1.011$. With such a configuration, set S remains linear independent (and therewith M has rank n) with probability $1 - O(\frac{1}{n})$. Moreover, it gives reasonable real-world performance: 5 calls to a PRG and accesses to the table per $\mathsf{addItem}$ and a small storage overhead of only 1%.

B. δ-Recovery and Choice of c and k

More challenging is to show that the rank of matrix M remains equal to n, as long as $c = \frac{m}{n}$ is greater than a threshold c_0, and the number δ of rows removed from M is bound by $\sqrt{n}$. Following Definition 3, full rank despite removal of $\sqrt{n}$ rows automatically implies that Π provides $\sqrt{n}$-recovery. This is our main result and stated in Corollary 1.

While it is possible to show the existence of a phase transition phenomena (i.e., a necessary and sufficient condition for being able to decode all data items), in this paper we focus on a bound of $c = \frac{m}{n}$ that guarantees recovery (decodablity). The proof technique follows Calkin's proof for analyzing the rank of a binary matrix with columns of constant weight [8].

We extend the proof technique to the case of columns with hypergeometrically distributed weight. The hypergeometric distribution is the result of the random deletion of δ rows from the original matrix.

The proof outline is as follows. First, we consider Markov chain MH defined by the random walk on the hypercube 2^m using vectors of hypergeometric weight. In Theorem 2, we show that the expected rank of matrix M directly derives from the eigenvalues of MH's transition matrix (called H). We establish bounds for the eigenvalues of H, for the considered values of k, δ, and c, that lead to the asymptotic rank guarantee. In this section, k refers to the constant weight of the generator matrix column vectors, i.e., the number of distinct cells each data item gets XORed to. At a later stage, we will set $k = 5$.

Let $S_{m,k,\delta}$ denote the set of vectors over $GF(2)$ of length m and Hamming weight $k-\kappa$, where κ is hypergeometrically distributed, i.e.,

$$Pr[\text{weight}(u \in S_{m,k,\delta}) = k - \kappa] = \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}}.$$

M can be viewed as a matrix of n columns $u_1, u_2, \ldots, u_n$, chosen randomly from $S_{m,k,\delta}$. Let r be the rank of M, and the difference between n and rank r is $d = n - r$. We write $E(\cdot)$ for the expectation of a random variable. The main result, Corollary 1, directly derives from the following Theorem 1.

Theorem 1. If $\delta < \sqrt{n}$, then there exists $c_0 > 1$ such that

$$\text{if } c > c_0 \text{ and } m > c \cdot n \implies E(2^d) \to 1 \text{ as } m \to \infty.$$

Proof. This is the main theorem that we will prove in several steps below. □

Corollary 1 (δ-Recovery). If $\delta < \sqrt{n}$, then there exists $c_0 > 1$ such that

$$\text{if } c > c_0 \text{ and } m > c \cdot n$$
$$\implies$$
$$Pr[\text{rank}(M) = n] = 1 - o(1).$$

Proof. This derives immediately from Theorem 1, since $E(2^d) \to 1$ (when $n \to \infty$) implies that $d \to 0$, and therefore the rank $r \to n$.

Finally, if matrix M has rank n, then we recover all n data items with Gaussian elimination, which results in δ-recovery. □

We now turn to Theorem 1. To prove this theorem, we first and similarly to the analysis for the case of u_i having constant weight [8] define a random walk on the 2^m hypercube using steps u_i of hypergeometrically distributed weight. Let the random variable describing the position on the hypercube be x_i.

$$x_0 = 0 \text{ and } x_i = x_{i-1} + u_i$$

We introduce the Markov chain MH with state defined by the weight of x_i.

Lemma 2. The transition matrix H of MH has the following two properties:

1) $H = \sum_{\kappa=0}^{k} Pr[\text{weight}(u_i) = \kappa] \cdot A^{(\kappa)}$, where $A^{(\kappa)}$ is the transition matrix for the random walk Markov chain given by u_i of constant weight κ.
2) $H_{(p,q)} = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}}$, for $0 \leq p, q \leq k$ and where the binomial coefficients are interpreted to be 0, if $\kappa + p + q$ is odd.

Proof. $H_{(p,q)}$ denotes the probability of transitioning from state p to state q. Note that here, when we are in state p, we add a random vector u_i of hypergeometrically distributed weight k. Therefore, $H_{(p,q)}$ is basically the sum of the probability of u_i having a given weight κ *times* the probability that we transition from state p to state q with fixed weight κ (which is by definition $A^{(\kappa)}_{(p,q)}$ and is equal to $\frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}}$).

$$\begin{aligned} H_{(p,q)} &= \sum_{\kappa=0}^{k} Pr[\text{weight of } u_i \text{ is } \kappa] \cdot A^{(\kappa)}_{(p,q)} \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} A^{(\kappa)}_{(p,q)} \qquad (1) \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{\binom{q}{\frac{\kappa-p+q}{2}}\binom{m-q}{\frac{\kappa+p-q}{2}}}{\binom{m}{\kappa}} \end{aligned}$$

Equation 1 also directly implies $H = \sum_{\kappa=0}^{k} Pr[\text{weight}(u_i) = \kappa] \cdot A^{(\kappa)}$. □

Lemma 3. H's eigenvalues λ^H_i are a linear combination of the eigenvalues $\lambda_{\kappa,i}$ of the constant weight (κ) transition matrices. Formally,

$$\lambda^H_i = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{n-k}{\delta-k+\kappa}}{\binom{n}{\delta}} \lambda_{\kappa,i} \qquad (2)$$

$$\lambda_{\kappa,i} = \sum_{t=0}^{\kappa} (-1)^t \frac{\binom{i}{t}\binom{m-i}{\kappa-t}}{\binom{m}{\kappa}} \qquad (3)$$

$$e_i[j] = \sum_{t=0}^{j} (-1)^t \binom{i}{t}\binom{m-i}{j-t} \qquad (4)$$

Proof. From [8] (Lemma 2.2), $A^{(\kappa)} = \frac{1}{2^m} U \Delta^{(\kappa)} U$, where U is defined by columns (eigenvectors) e_i, and Δ is the diagonal eigenvalues matrix defined by λ^H_i. We also note the following property:

$$U^2 = 2^n I. \qquad (5)$$

Then,

$$\begin{aligned} H &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} A^{(\kappa)} \\ &= \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \frac{1}{2^m} U\Delta^{(\kappa)}U \\ &= \frac{1}{2^m} U(\sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \cdot \Delta^{(\kappa)})U \\ &= \frac{1}{2^m} U\Lambda U \end{aligned}$$

where $\Lambda = \sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{m-k}{\delta-k+\kappa}}{\binom{m}{\delta}} \cdot \Delta^{(\kappa)}$. Given that $\Delta^{(\kappa)}$ is diagonal, Λ is also diagonal, and therefore $\frac{1}{2^m}U\Lambda U$ is the eigen-decomposition of H with eigenvectors the columns of U and eigenvalues $\sum_{\kappa=0}^{k} \frac{\binom{k}{k-\kappa}\binom{n-k}{\delta-k+\kappa}}{\binom{n}{\delta}} \lambda_{\kappa,i}$. □

Remark. The eigenvectors of H do not depend on κ and form matrix U. It therefore does not matter if we take a step of size z and then z' or z' and then z. The probability that $u_1, u_2, \ldots, u_t$ sum to 0 is the 00^{th} coefficient of H^t. Therefore, considering all possible combinations of t columns of u_i, and given the fact that we are operating in $GF(2)$, the expected number of combinations of u_i that add-up to 0, we derive

$$E(2^d) = \sum_{t=0}^{m} \binom{m}{t} (H^t)_{00}.$$

Theorem 2.

$$E(2^d) = \sum_{t=0}^{m} \frac{1}{2^m} \binom{m}{i} (1 + \lambda_i^H)^n$$

Proof. We already know that $H = \frac{1}{2^m}U\Lambda U$. We can therefore derive H^t:

$$H^t = (\frac{1}{2^m}U\Lambda U)^t = \frac{1}{2^m}U\Lambda^t U \tag{6}$$

Equation 6 derives from equation 5, which states that $U \cdot U = 2^n I$.

Given that U is defined by the eigenvectors e_i, the 00^{th} coefficient of H^t can be calculated as:

$$(H^t)_{00} = \sum_{i=0}^{m} \frac{1}{2^m} (\lambda_i^H)^t \binom{m}{i}$$

Since u_i are chosen randomly and independently, the expected number of subsequences that add up to 0 is:

$$\begin{aligned} E(2^d) &= \sum_{t=0}^{n} \binom{n}{t} (H^t)_{00} \\ &= \sum_{t=0}^{n} \binom{n}{t} \sum_{i=0}^{m} \frac{1}{2^m} (\lambda_i^H)^t \binom{m}{i} \\ &= \sum_{i=0}^{m} \frac{1}{2^m} \binom{m}{i} (1 + \lambda_i^H)^n \end{aligned}$$

□

Lemma 4. H's eigenvalues satisfy the following:

1) $\forall 0 \le i \le n$: $|\lambda_i^H| < 1$
2) $\forall t < \frac{1}{2}$, if $i = t \cdot m$, then

$$\lambda_i^H < (1-\frac{2i}{m})^3 - \frac{4\binom{k}{2}}{m}(1-\frac{2i}{m})\frac{i}{m}(1-\frac{i}{m}) + O(\frac{k^3}{c^2m^2}) + O((\frac{\delta}{m})^3)$$

3) For i sub-linearly close to $\frac{m}{2}$, i.e., $\frac{m}{2} - i = \frac{m^\theta}{2}$ (for some $\theta < 1$), we have

$$\lambda_i^H < (\frac{1}{m^{1-\theta}})^3 - \frac{4\binom{k}{2}}{m}(\frac{1}{m^{1-\theta}}) + O(\frac{k^3}{m^2}) + O((\frac{\delta}{m})^3) \tag{7}$$

Proof. The first inequality derives from the formulae of λ_i^H, and $\lambda_{k,i}$ as shown in Lemma 3, equations 2 and 3. The second inequality derives from the bound of $\lambda_{\kappa,i}$ as derived by Lemma 3.1(c) in [8], combined with Equation 2 setting k to 5. The last inequality derives from replacing $\frac{m}{2} - i$ by $\frac{m^\theta}{2}$. □

Corollary 2. For $\delta < m^\gamma$ where $\gamma < \frac{2}{3}$, if $\theta < \frac{2}{3}$ and $\frac{m}{2} - i = \frac{m^\theta}{2}$, then $\lambda_i^H m \to 0$ as $m \to 0$.

Proof. This derives from Equation 7. □

This is in particular true for $\gamma = \frac{1}{2}$, and we therefore set $\delta = \sqrt{n}$.

Lemma 5. There exists $c_0 > 1$ such that if $c > c_0$ and $m > c \cdot n$ then $\exists \epsilon$ such that:

1) $A = \sum_{i=0}^{\epsilon m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n + \sum_{i=(1-\epsilon)m}^{m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 0$ as $m \to +\infty$
2) $B = \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 1$ as $m \to +\infty$
3) $C = \sum_{i=\frac{m}{2}(1-\epsilon)}^{\frac{m}{2}-m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n + \sum_{i=\frac{m}{2}+m^{\frac{4}{7}}}^{\frac{m}{2}(1+\epsilon)} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 0$ as $m \to +\infty$
4) $D = \sum_{i=\epsilon m}^{\frac{m}{2}(1-\epsilon)} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n + \sum_{i=\frac{m}{2}(1+\epsilon)}^{(1-\epsilon)m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 0$ as $m \to +\infty$

Proof. Since the eigenvalues λ^H of H satisfy the same asymptotic bounds as λ_i as in the case of [8], the results derive similarly. Note that (1), (2), and (3) are always true independently of c_0. (4) is also, but requires c_0 to be a solution to a function f defined in the next steps.

1) Given that $|\lambda_i^H| < 1$,

$$\begin{aligned} A &= \sum_{i=0}^{\epsilon m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \\ &\quad + \sum_{i=(1-\epsilon)m}^{m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \\ &< \sum_{i=0}^{\epsilon m} \frac{1}{2^{m-n-1}}\binom{m}{i}. \end{aligned}$$

Therefore, $A \to 0$ as $m \to \infty$.

2) When, $\frac{m}{2} - m^{\frac{4}{7}} \leq i \leq \frac{m}{2} + m^{\frac{4}{7}}$, we have $(1 + \lambda_i^H)^n = (1 + o(\frac{1}{m}))^n = 1 + o(1)$. Therefore, $B \approx \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i} \to 1$, as $m \to \infty$.

3) For $\frac{m}{2}(1-\epsilon) \leq i \leq \frac{m}{2} - m^{\frac{4}{7}}$, we get $\lambda_i^H < \lambda^H_{\frac{n}{2}(1-\epsilon)}$. Setting $\epsilon = \frac{1}{m^{1-\theta}}$ (therefore $\epsilon = o(n^{-3})$, $\delta = \sqrt{m}$, and combining with Equation 7, we obtain $\lambda_i^H < \epsilon^3 - \frac{\binom{k}{2}}{m}\epsilon + O(\frac{1}{m^{\frac{3}{2}}})$.
Thus,

$$\begin{aligned}(1+\lambda_i^H)^n < e^{m\epsilon^3} e^{-\frac{\binom{k}{2}}{m}\epsilon} &\to 1 \\ m &\to +\infty \text{ (for } \theta < \frac{2}{3})\end{aligned}$$

Also, we can bound binomial term in the sum as follows:

$$\begin{aligned}\binom{m}{i} &< (\frac{me}{\frac{m}{2} - m^{\frac{4}{7}}})^{\frac{m}{2}-m^{\frac{4}{7}}} = (\frac{2e}{1-2m^{\frac{-3}{7}}})^{\frac{n}{2}-m^{\frac{4}{7}}} \\ &< (2e)^{\frac{n}{2}-m^{\frac{4}{7}}}\end{aligned}$$

Therefore, $\frac{1}{2^m}\binom{m}{i} < \frac{(2\cdot e)^{\frac{n}{2}-m^{\frac{4}{7}}}}{2^m} < (2\cdot e)^{\frac{-m}{14}}$.
Finally,

$$\begin{aligned}B &= \sum_{i=\frac{m}{2}-m^{\frac{4}{7}}}^{i=\frac{m}{2}+m^{\frac{4}{7}}} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 0 \\ &< 2m^{\frac{4}{7}}(2e)^{\frac{-m}{14}} \to 0 \text{ as } m \to +\infty\end{aligned}$$

4) For the last term

$$\begin{aligned}D = &\sum_{i=\epsilon m}^{\frac{m}{2}(1-\epsilon)} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \\ &+ \sum_{i=\frac{m}{2}(1+\epsilon)}^{(1-\epsilon)m} \frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \to 0,\end{aligned}$$

by symmetry we consider the first term. Let $\alpha = \frac{i}{m}$ and $\beta = \frac{n}{m}$. We have [12]:

$$\begin{aligned}\binom{m}{i} &< 2^{mH(\frac{i}{m})} = e^{m\log(2)H(\alpha)} \\ &= e^{m(-\alpha\log(\alpha)-(1-\alpha)\log(1-\alpha))}.\end{aligned}$$

Furthermore, from Lemma 4, we have $\lambda_i^H < (1-\frac{2i}{m})^3 = (1-2\alpha)^3$. Therefore,

$$\begin{aligned}&\frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n \\ &< e^{m(-\log 2-\alpha\log(\alpha)-(1-\alpha)\log(1-\alpha)+\beta\log(1+(1-2\alpha)^3))}\end{aligned} \quad (8)$$

Looking at the exponent in Inequality (8), we now define a function f which will determine the threshold value c_0. This threshold value is also the expansion threshold used in Theorem 1. Let $f(\alpha, \beta) = -\log 2 - \alpha\log(\alpha) - (1-\alpha)\log(1-\alpha) + \beta\log(1+(1-2\alpha)^3)$. Therefore,

$$\frac{1}{2^m}\binom{m}{i}(1+\lambda_i^H)^n < e^{mf(\alpha,\beta)}.$$

We need to find a value β_0, such that for all α (defined as $\frac{i}{m}$) $\in (\epsilon, 1-\epsilon)$, we guarantee that $f(\alpha, \beta) < 0$. Such value β_0 is found by considering (α_0, β_0) the root of

$$\begin{aligned}f(\alpha, \beta) &= 0 \\ \frac{\partial f(\alpha,\beta)}{\partial \alpha} &= 0.\end{aligned}$$

Setting $c > c_0$, we get $\beta < \beta_0$, and therefore $D \to 0$ as $m \to +\infty$.

Solving $f(\alpha, \beta) = 0$ for β gives $\beta = (\alpha \cdot \log(\alpha) - \alpha \cdot \log(-\alpha+1) + \log(2) + \log(-\alpha+1)) / \log(-(2\cdot\alpha-1)^3 + 1)$. Using SageMath, we then numerically approximate the minimum β_0 for $0 < \alpha < \frac{1}{2}$ and obtain $\beta_0 = 0.88949$ which means $c_0 = 1.1243$. □

Proof of Theorem 1. For $\delta < \sqrt{n}$, let $c_0 = \frac{1}{\beta_0}$. From Lemma 5, we get

$$\text{if } c > c_0 \text{ and } m > c\cdot n \implies E(2^d) = A+B+C+D \to 1 \text{ as } m \to \infty.$$

□

C. Computational Optimization of Recovery

Our listItems in Algorithm 3 solves a system of linear equations using Gaussian elimination, see Line 27. However, vector $\vec{c}$ is a vector over field $(GF(2^{\ell+2\cdot s}))^n$. For many real-world values of plaintext length ℓ, field sizes become very large. For example, for syslog events of maximum length 1024 byte [16], 128 bit random counters, and 256 bit HMAC-SHA2, operations must be in $GF(2^{8576})$ which turns out to be a problem. Gaussian elimination over large fields is extremely slow in practice even when using modern, optimized computer algebra systems. To mitigate the problem, we exploit a special property of our addItem algorithm. Note that XOR operations during addItem and M's coefficients of either 0 or 1 make M a (sparse) matrix over $GF(2)$. Thus, we use the following tweak to speed up solving.

Instead of solving $M \cdot \vec{x} = \vec{v}$ over $(GF(2^{\ell+2\cdot s}))^n$, we compute a reduced row echelon form E of M together with a $m \times m$ transformation matrix T such that $T \cdot M = E$. As all operations are over $GF(2)$, computations of E and T are fast, see the evaluation in Section VI. As $M \cdot \vec{x} = \vec{v}$, we multiply with T from the left and get $T \cdot M \cdot \vec{x} = E \cdot \vec{x} = T \cdot \vec{v}$. That is, we convert T from $GF(2)$ into $(GF(2^{\ell+2\cdot s}))^n$ and multiply T with $\vec{v}$ to get $\vec{x}$. T's conversion is cheap, and instead of cubic complexity for Gaussian elimination over a large field, multiplying $m \times m$ matrix T with vector $\vec{v}$ has only quadratic complexity in m.

As a result, Gaussian elimination becomes significantly faster. To illustrate, computing the rank of a small size 500×500 random matrix over $GF(2)$ takes roughly 3 ms in SageMath on a 2.2 GHz mobile Intel Skylake i7 CPU.

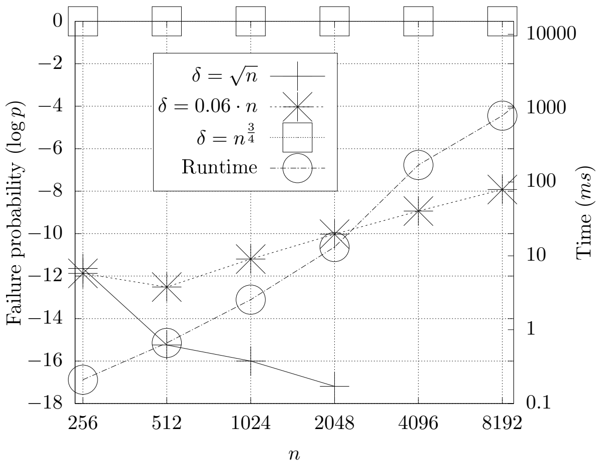

Fig. 3: Failure probability and runtime of listItems

Computing the rank of the exact same matrix converted to $GF(2^{8576})$ takes about 12 s. In conclusion, this optimization results in a speed-up of roughly two orders of magnitude.

VI. Experimental Analysis

To back up our theoretical claims, we have also performed a practical analysis of our coding technique. The goal of this analysis is to estimate and give confidence in the probability that listItems recovers all data for a concrete choice of parameters k, c, n, and δ following our theoretical prediction in Section V-B.

We have implemented our coding and decoding techniques in C, and the implementation is available for download [1]. Our implementation first builds a random binary $(m = c{\cdot}n) \times n$ matrix M where each column has weight k. Therewith, we simulate encoding (and addItem) for a total of n data items. To simulate adversarial behavior or a crash, δ rows are randomly chosen and removed from M by setting all coefficients to 0. To simulate decoding (and listItems), the implementation computes standard Gaussian elimination to convert M into reduced row echelon form and derives M's rank.

As (our) standard Gaussian elimination is asymptotically expensive with $O(n^3)$ computational complexity, the implementation features two performance improvements. First, M is a binary matrix and Gauss' operations are only XOR of rows. So, we use Intel's AVX2 SIMD instructions, store rows in 256 bit pieces, and XOR each pair of pieces with only one CPU instruction. Our second performance improvement integrates OpenMP parallel processing and parallelizes the inner loop of Gaussian elimination. Especially for larger n, parallelization gives performance improvements. We also stress that, in our case of sparse matrices M, Gaussian elimination allows for significant performance improvements [27] which we leave to future work.

We run our experiments with $k = 5$ and $c = 1.1244$, as suggested by Section V-B, for $n \in \{256, 512, 1024, 2048, 4096, 8192\}$ and $\delta = \sqrt{n}$. Therewith, we want to indicate that our theory of recovering up to $\sqrt{n}$ lost or corrupted entries in M can be recovered with high probability. We additionally run experiments with $\delta = n^{\frac{3}{4}}$ and even $\delta = 0.06 \cdot n\}$, i.e., a small, but constant fraction of n. The idea here is to show that, as predicted, our coding cannot recover from values of δ significantly larger than $\sqrt{n}$. For each combination of parameters, we perform 2^{20} repetitions and compute $p = \frac{\#(M \text{ has rank} < n)}{2^{20}}$.

Figure 3 shows the outcome of our experiments. The x-axis (log scaled) shows the number of data items, and the y-axis shows $\log p$. For $\delta = \sqrt{n}$ and both $n = 4096$ and $n = 8192$, we have $p = 0$, i.e., all 2^{20} random matrices M had full rank n. For $\delta = n^{\frac{3}{4}}$, all M had rank less than n. The same holds for $\delta = 0.06 \cdot n$. These results are consistent with our theoretical asymptotic bounds. Starting with relatively low values of n (e.g., $n = 4096$), the system does not experience any failures, indicating that our prediction of setting $\delta < \sqrt{n}$ to decode with high probability is sound. In contrast, increasing δ significantly beyond $\sqrt{n}$ jeopardizes decodability. Figure 3 also shows average listItems runtime on a 2.2 GHz i7-6560U CPU. The runtime scales closely to the anticipated cubic increase of Gaussian elimination. Again, we expect a real-world implementation, leveraging sparsity of matrices, to improve complexity down to quadratic.

Note that our security setup is different from traditional setups: in each run the adversary would only get one chance to tamper with the system log. Therefore, in 2^{20} experiments, not a single time was the adversary able to delete δ entries that the system could not recover. The experimental results also confirm that the proposed coding scheme exhibits a threshold phenomena, as for $\delta \geq n^{\frac{3}{4}}$ the recovery fails with high probability, supporting the claim of near-optimality.

VII. Conclusion

We have presented a new data structure Π together with several new mechanisms combining forward integrity with data recovery. At the center of our techniques lies an efficient, customizable coding scheme which provides high decodability guarantees even when a large number of data items are lost or maliciously modified. This coding scheme is online, requires a constant number of writes and operation per input symbol, and therefore enables the integration of forward security mechanisms. Our formal analysis and practical experiments show that for any number n of log items, a space overhead of only 12% and a computational overhead of a factor of 5 (in terms of write operations) is sufficient to decode and recover all log items with high probability $1-o(1)$. The coding scheme is of independent interest on its own.

While secure and robust audit data storage is a prime application for our techniques, one can envision other applications. Whenever an adversary threatens to compromise a system, tamper with stored data, and tries hiding traces to avoid detection, our techniques will be useful.

References

[1] Anonymous. Source code for experiments, 2019. https://github.com/QuickGauss/QuickGaussImplementation.

[2] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010. ISSN 0933-2790.

[3] M. Bellare and B.S. Yee. Forward integrity for secure audit logs. Technical report, UC San Diego, 1997. https://pdfs.semanticscholar.org/0885/7824495937ed6f3dbb5b4d34e0481a73aef8.pdf.

[4] M. Bellare and B.S. Yee. Forward-Security in Private-Key Cryptography. In *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, pages 1–18, 2003.

[5] E.-O. Blass and G. Noubir. Secure Logging with Crash Tolerance. In *Conference on Communications and Network Security*, pages 1–10, Las Vegas, USA, 2017.

[6] K.D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, pages 46–67, 2014.

[7] N.J. Calkin. Dependent sets of constant weight vectors in $GF(q)$. *Random Struct. Algorithms*, 9(1-2):49–53, 1996.

[8] N.J. Calkin. Dependent Sets of Constant Weight Binary Vectors. *Combinatorics, Probability & Computing*, 6(3): 263–271, 1997.

[9] T.M. Cover and J.A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience, 2006.

[10] M. Dietzfelbinger and R. Pagh. Succinct Data Structures for Retrieval and Approximate Membership (Extended Abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 385–396, 2008.

[11] P. Erdős and A. Rényi. On the Evolution of Random Graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[12] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[13] R.G. Gallager. Low-density parity-check codes. *IRE Trans. Information Theory*, 8(1):21–28, 1962.

[14] M.T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. In *Allerton Conference on Communi-*

[17] D. Ma and G. Tsudik. A New Approach to Secure

cation, Control, and Computing, pages 792–799, Monticello, USA, 2011.

[15] J.E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Australasian Symposium on Grid Computing and e-Research*, volume 54, pages 203–211, 2006.

[16] Chris M. Lonvick. The BSD Syslog Protocol. RFC 3164, August 2001. URL https://rfc-editor.org/rfc/rfc3164.txt.

Logging. *ACM Transactions on Storage*, 5(1), 2009. ISSN: 1553-3077.

[18] G.A. Marson and B. Poettering. Practical Secure Logging: Seekable Sequential Key Generators. In *ESORICS*, pages 111–128, 2013.

[19] G.A. Marson and B. Poettering. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *ESORICS*, pages 37–54, 2014.

[20] M. Mitzenmacher. Digital fountains: a survey and look forward. In *Information Theory Workshop*, pages 271–276, Oct 2004.

[21] S. Pontarelli, P. Reviriego, and M. Mitzenmacher. Improving the performance of Invertible Bloom Lookup Tables. *Inf. Process. Lett.*, 114(4):185–191, 2014.

[22] T. J. Richardson and R. L. Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, 47(2):638–656, Feb 2001.

[23] P. Rogaway. Nonce-Based Symmetric Encryption. In *Proceedings of FSE*, pages 348–359, Delhi, India, 2004. ISBN 3-540-22171-9.

[24] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[25] C.E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[26] A. Shokrollahi. *LDPC Codes: An Introduction*, volume 23 of *Coding, Cryptography and Combinatorics*, pages 85–110. Birkhäuser, 2004. ISBN 978-3-0348-9602-3.

[27] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, 32(1):54–62, January 1986. ISSN 0018-9448.

[28] A.A Yavuz, P. Ning, and M.K. Reiter. BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems. *Transactions on Information System Security*, 15(2):9, 2012. ISSN 1094-9224.